Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#70: Fast Graphics Hints

Written by: Don Marsh & Jim Luther September 1989

This Technical Note discusses techniques for fast animation on the Apple IIGS.

QuickDraw II gives programmers a very generalized way to draw something to the Super Hi-Res screen or to other parts of Apple IIGS memory. Unfortunately, the overhead in QuickDraw II makes it an unacceptable tool for all but simple animations. If you bypass QuickDraw II, your application has to write pixel data directly to the Super Hi-Res graphics display buffer. It also has to control the `New-Video` register at $C029, and set up the scan-line control bytes and color palettes in the graphics display buffer. Chapter 4 of the *Apple IIGS Hardware Reference* documents where you can find the graphics display buffer in memory and how the scan-line control bytes, color palettes, and pixel data bytes are used in Super Hi-Res graphics mode. The techniques described in this Note should be used with discretion—we do **not** recommend bypassing the Apple IIGS Toolbox unless it is absolutely necessary.

Map the Stack Onto Video Memory

To achieve the fastest screen updates possible, you must remove all unnecessary overhead from the instructions that perform graphics memory writes. The obvious method for achieving sequential writes to the graphics memory uses an index register, which must be incremented or decremented between writes. These operations can be avoided by using the stack. Each time a byte or word is pushed onto the stack, the stack pointer is automatically decremented by the appropriate amount. This is faster than doing an indexed store followed by a decrement instruction.

But how is the stack mapped onto the graphics memory? The stack can be located in bank $01 instead of bank $00 by writing to the `WrCardRAM` auxiliary-memory select switch at $C005. Bank $01 is shadowed into $E1 by clearing bit 3 of the `Shadow` register at $C035. Under these conditions, if the stack pointer is set to $3000, the next byte pushed onto the stack is written to $013000, then shadowed into $E13000. The stack pointer is automatically decremented so the stage is set for another byte to be written at $E12FFF.

Warning: While the stack is mapped into bank $01, you may **not** call **any** firmware, toolbox or operating system routines (ProDOS 8 or GS/OS). Don't even think about it.

Unroll All Loops

Another source of overhead is branching instructions in loops. By “straight-lining” the code to move up a scan-line’s worth of memory at one time, branch instructions are avoided. Following is an example of this technique.

```
lda    |164,y              ; accumulator is 16 bits for
pha                        ; best efficiency
lda    |162,y
pha
lda    |160,y
pha
```

In this example, the `Y` register is used to point to data to be moved to the graphics memory, and hard-coded offsets from the `Y` register are used to avoid register operations between writes.

Hard-Code Instructions and Data

In desperate circumstances, it is necessary to remove overhead from the previous code example. This can be accomplished by hard-coding pixel data into your code instead of loading pixel values from a separate data space and transferring them to the graphics memory (as in the example). If you are writing an arbitrary pattern of three or fewer constant values to the screen, for example, the following method is the fastest known:

```
lda    #val1
ldx    #val2
ldy    #val3
pha                        ; arbitrary pattern of pushes
phx
phy
phy
phx
```

In cases where many different values must be written to the screen, pixel data can be written to the screen using immediate push instructions:

```
pea    $5389               ; some arbitrary pixel values
pea    $2378
pea    $A3C1
pea    $39AF
```

Your program can generate this mixture of `PEA` instructions and pixel data itself, or it could load pixel data that already has `PEA` instructions intermixed (thus increasing the data size by one half).

Be Aware of Slow-Side and Fast-Side Synchronization

Estimating execution speed by counting instruction cycles is always a challenging task on the IIGS, but it is particularly tricky when one is writing to the graphics memory. The graphics memory resides in the side of the IIGS system controlled by the 1 MHz Mega II chip, which means that during all writes to this memory, the fast side of the system controlled by the Fast Processor Interface (FPI) chip must be synchronized with slow side of the system controlled by the Mega II, even if the system is running code at full native speed. This synchronization is performed automatically and transparently by the FPI in the IIGS, and it isn't normally of concern to the programmer. Animation programmers must worry about synchronization delays, however, because slight changes in graphics update code may change the frequency of these delays, and hence the speed of the program. In practical terms, this means that one loop writing

data to the graphics memory may run at the same speed as a second loop with a higher cycle count.

A careful analysis of the synchronization problem leads to the following tables, which are useful as a rough estimate of the speed attained by different pieces of code. Each entry is based on the number of cycles consumed during consecutive write instructions. For example, a series of `PEA` instructions requires five cycles for each 16-bit write. A short `PHA` instruction followed by a branch requires six cycles for each 8-bit write.

Fast Cycles per Write (byte)	Actual Speed (*µ*sec./byte)
3 to 5	2.0
6 to 8	3.0
9 to 11	4.0

Fast Cycles per Write (word)	Actual Speed (*µ*sec./word)
4 to 6	3.0
7 to 8	4.0
9 to 11	5.0

The times given in the tables apply only if the same number of fast cycles separate each consecutive write operation. The first write operation in a set of write instructions usually takes longer than subsequent writes, because the potentially long synchronization operation is accomplished at that time. Unpredictable delays caused by memory refresh slow things down further, although refresh delays byte-wide writes more often than word-wide writes. Therefore, it is usually preferable from a speed standpoint to use word-wide writes to the graphics memory.

For more information on synchronization cycle timing within the IIGS, see Chapter 2 of the *Apple IIGS Hardware Reference* and Apple IIGS Technical Note #68, Tips for I/O Expansion Slot Card Design.

Use Change Lists

The timing data given in the preceding section shows that it is not possible to perform full-screen updates in the time it takes the IIGS to scan the entire screen. In fact, it would be difficult to update more than one-sixth of the screen in one scan time. Therefore, it is necessary to update only those pixels which have actually changed from the previous frame of animation. One method of doing this is to precalculate the pixels which change by comparing each frame against the preceding frame. For interactive animation, fast methods must be developed for predicting which areas of the screen must be updated (a determination of the exact pixels might require more computation than the actual update would require).

Using the Video Counters

To achieve “tear-free” screen updates, it is necessary to monitor the location of the scan-line beam when writing to graphics memory. As described in Apple IIGS Technical Note #39, Mega II Video Counters, the `VertCnt` and `HorizCnt` Mega II video counter registers at $C02E-C02F allow you to determine which scan line is currently being drawn.

By using only the `VertCnt` register and ignoring the low bit of the 9-bit vertical counter stored in `HorizCnt`, you can determine within 2 scan lines which scan line is currently being drawn. The `VertCnt` video counter contains the number of the current scan line divided by two, offset by $80. For example, if the scan-line beam was currently refreshing either scan line four or five, `VertCnt` would contain $82 (4/2 + $80 or 5/2 + $80). Vertical blanking happens during `VertCnt` values $7D through $7F and $E4 through $FF.

Clever updates can modify twice as many pixels on the screen by sacrificing some smoothness, running at 30 frames per second instead of 60. The technique is as follows:

1. Wait for the scan line beam to reach the first scan line.
2. Start updates from the top of the screen, being careful not to pass the scan line beam.
3. Continue updates while the scan line beam progresses toward the bottom of the screen, then goes into vertical blanking, then restarts at the top of the screen.
4. Finish the update before the scan line beam catches the update point.

Careful use of this method allows a frame to be updated during two scans of the screen instead of just one. If you are not sufficiently careful, tearing results.

Note: The Apple IIGS main logic board Mega II-VGC registers and interrupts are not synchronous to the Apple II Video Overlay Card video and therefore should not be used for time synchronization with the Apple II Video Overlay Card video output. However, they can be used for time synchronization with the Apple IIGS video output. See the *Apple II Video Overlay Card Development Kit* for more information.

Interrupts

It is not possible to support interrupts while sustaining a high graphics update rate, unless jerkiness or tearing is acceptable. Be aware that many system activities such as GS/OS and AppleTalk depend on interrupts and do not function if interrupts are disabled.

Further Reference

- *Apple IIGS Firmware Reference*
- *Apple IIGS Hardware Reference*
- *Apple II Video Overlay Card Development Kit*
- Apple IIGS Technical Note #39, Mega II Video Counters
- Apple IIGS Technical Note #40, VBL Signal
- Apple IIGS Technical Note #68, Tips for I/O Expansion Slot Card Design